文字列検索における圧縮インデックス構築の省メモリな並列化手法

林　伸也† 　田浦　健次朗†

文字列検索はテキストを扱うアプリケーションの基礎を成す重要な技術である．テキストの任意部分文字列の検索を可能にするインデックスとして suffix array があるが, 元のテキストに比べてサイズが非常に大きくなるという問題がある．そこで, suffix array と同様の機能を持ちながら, より小さいメモリで動作する圧縮インデックスに注目が集まっている．本研究では, そのような圧縮インデックスの一つである FM-Index を構築する上で, 時間的・空間的に支配的な Burrows-Wheeler 変換の省メモリな並列化手法を提案する．

Lightweight parallel index construction for string matching

Shinya Hayashi† and Kenjiro Taura†

String matching is a fundamental task in text processing. Though suffix array is a very useful index data structure for string matching, its space usage is problematic. Compressed index, which provides almost the same functionality as suffix array and can be expressed in succinct form, is a valid solution for the space problem of suffix array. In this paper, we propose a lightweight and parallel method to conduct a transformation called BWT, which is a bottleneck in constructing compressed index. a transformation called BWT, a dominant phase in constructing compressed index.

1. はじめに

テキストから有用な情報を抽出するときにはさまざまな技術が必要となるが, 特に文字列検索はそれらのアプリケーションの基礎を成す重要な処理である．文字列検索のための古典的なインデックスである suffix array[7] はテキストよりも多くの容量を消費する．

この問題の解決策として, インデックスの圧縮が有効である．圧縮インデックスの中でも, FM-Index[3] と呼ばれるものはインデックスが元のテキストを復元するのに十分な情報を含んだ形で圧縮されており, 多くの研究が行われている[5][9][4]．

このように非常に有用な性質を持った FM-Index であるが, その効率的な構築方法が問題となる． FM-Index を構築するときには, まずテキストに対して Burrows-Wheeler 変換 (BWT)[1] という変換を行う必要がある．BWT を定義通りに行うためには suffix array を構築する必要があるが, これでは自己インデックス構築にはワーキングスペースとして $O(n \log n)$ のメモリが必要となってしまい, メモリ使用量を小さくするという圧縮の目的が達成されない．

BWT の省メモリな実現方法はいくつか研究が行われている．その中でも岡野原らは $O(n)$ という時間計算量を達成しており, これは非常に高速である．しかし, テキストが大規模になってくると, 並列化によって $O(n)$ よりも速い計算時間を達成することが求められる．しかし, 岡野原らの手法はテキストをシーケンシャルに処理していくため, 並列化を行うことは難しい．Suffix array の構築は並列化が行われており[6], また suffix array さえ構築できれば BWT を並列に行うのは容易である．しかし, このような方法では $O(n \log n)$ のメモリ消費は避けられない．

そこで本研究では, メモリ上での省メモリかつ並列化可能な BWT の構築アルゴリズムを提案する．

1.1 記号の定義

テキスト T は文字の並びから成る．各文字はアルファベットと呼ばれる集合の要素である．アルファベット集合のサイズは σ である．T の i 番目の文字を T_i または $T[i]$ で表す．添字は 0 から始まることとする．また, T のサイズは $|T| = n + 1$ であるとする．ただし, $T_n = \$$ である．$は番兵で, アルファベットのどの文字よりも辞書順で小さい文字であるとする．T の i 文字目から j 文字目までの部分文字列を $T[i, j]$ と書く．

† 東京大学
The University of Tokyo

ⓒ 2013 Information Processing Society of Japan

		i	SA[i]		bwt
mississippi$		0	11	$mississippi	i
ississippi$m		1	10	i$mississipp	p
ssissippi$mi		2	7	ippi$mississ	s
sissippi$mis		3	4	issippi$miss	s
issippi$miss	sort	4	1	ississippi$m	m
ssippi$missi	→	5	0	mississippi$	$
sippi$missis		6	9	pi$mississip	p
ippi$mississ		7	8	ppi$mississi	i
ppi$mississi		8	6	sippi$missis	s
pi$mississip		9	3	sissippi$mis	s
i$mississipp		10	5	ssippi$missi	i
$mississippi		11	2	ssissippi$mi	i

図 **1** テキスト T=”mississippi$”の BWT
Fig. 1 BWT of a text T=”mississippi$”

2. 背 景 技 術

2.1 Suffix Array

Suffix array[7](SA) は T の全ての suffix を辞書順に並び替え, 各 suffix がテキスト中で出現する位置を格納した配列である. T の suffix とは, T の途中から始まり, 最後まで続く部分文字列を意味する. SA は任意の部分文字列に対するインデックスであるため, ゲノム配列やバイナリデータなど, 区切りが明確でないデータに対しても検索を行えるという特徴がある.

SA の空間計算量は $O(n \log n)$ ビットである. これは元のテキストに比べて $\log n / \log \sigma$ 倍大きい.

2.2 Burrows-Wheeler 変換

SA のスペースの問題を解決する手段として, 圧縮インデックスが有効である. 圧縮インデックスの中でも FM-Index[3] と呼ばれる手法では, まずテキストに対して Burrows-Wheeler 変換 (BWT)[1] を行う. 以下に BWT の定義を示す.

定義 1 (Burrows-Wheeler 変換)**.**

$$T_i^{bwt} = \begin{cases} T_n & (SA[i] = 0) \\ T_{SA[i]-1} & (otherwise) \end{cases} \qquad (1)$$

BWT により変換されたテキストには同じ文字が連続して現れやすいという性質がある. 例えば英語には”The”という単語が数多く出現するため, ”he”から始まる suffix の前に”T”という文字が現れる確率が高い. その結果, BWT の出力中で”he”から始まる suffix に対応する範囲には”T”が連続して並びやすくなる.[10] この性質により, BWT を行ったテキストは圧縮しやすくなる. 例として, $T =$ ”mississippi$” の BWT を図 **1** に示す. この例では $T^{bwt} =$ ”ipssm$pissii” となる.

以降では, BWT の出力として得られる文字列 T^{bwt} のことも BWT と呼ぶ.

2.3 LF-Mapping

ソートされた suffix の先頭文字を集めた配列を F, それらの 1 つ前の文字を集めた配列を L とする (L は BWT そのものである). このとき, L 中のある文字が F 中に現れる位置を求める操作のことを LF-mapping[1] という. LF-Mapping を求めるためには, 同じ種類の文字の並びは F, L において同じであるという性質を利用する. 以下に LF-mapping の定義を示す.

定義 2 (LF-Mapping)**.**

$$LF(i) = C(L_i) + rank(L_i, i) - 1 \qquad (2)$$

ただし $C(c)$ は T に現れる, 文字 c よりも小さい文字の数を返す関数, rank(c, i) は L の i 番目までに現れる文字 c の数を返す関数を表す. BWT$[i] = \$$ となる場所 i から, BWT$[i]$, BWT$[LF(i)]$, BWT$[LF^2(i)] \cdots$ というように LF-Mapping を繰り返すと, テキストを逆向きに復元することができる.

3. 関 連 研 究

3.1 サンプルソートを用いた方法

Kärkkäinen はサンプルソートを元にした手法を考案した. この手法では, まずいくつかの suffix をサンプリングし, ソートする. そして全ての suffix をサンプリングされた各 suffix と比較し, どの 2 つのサンプリングされた suffix の間に入るかで分類する. その後, 分類された各ブロックに対して順に SA の構築, BWT の計算を行う. こうすることで, テキスト全体に対する SA の構築を防ぎ, メモリ使用量を削減できる.

Kärkkäinen の手法は並列化が可能であるが, サンプリングする要素数を $r - 1$ としたとき, suffix の分類に $O(rn)$ の時間を要する. そのため, クリティカルパスが $O(n)$ を下回ることはできない.

3.2 BWT-IS

岡野原らは, SA の構築アルゴリズムである SA-IS[8] を BWT の構築に応用した BWT-IS を考案した. SA-IS ではまず各 suffix をテキスト中で一つ後ろの suffix との辞書的な大小に応じて L 型, S 型に分類する. さらに L 型に隣接する最左の S 型 suffix を LMS 型と定義する. 岡野原らの手法では, SA-IS において使用されるこれらの型の概念を BWT に対しても適用し, まず LMS 型の BWT を構築し, そこから L 型, S 型の BWT を順に計算する. このようにすることで, BWT を $O(n)$ の時間計算量で構築できる.

岡野原らの手法では, 例えばキューの要素を順に pop し, その値によって条件分岐し, 異なるキューに push する等, 実行順序を変更できない処理を含むため, 並列化を行うことは難しい.

ⓒ 2013 Information Processing Society of Japan

```
1  typedef struct BWT{
2    string bwt;
3    vector<int> sampledSA;
4    bit_sequence mark;
5  } BWT_t;
6
7  BWT_t Merge(const string &T, BWT_t &left,
         BWT_t &right){
8   BWT_t merged;
9   gap = ConstructGap(T, left, right);
10  merged.bwt = MergeBWT(T, left, right, gap);
11  merged.sampledSA = MergeSampledSA(T, left, right,
          gap);
12  merged.mark = MergeMark(T, left, right, gap);
13  return merged;
14 }
15
16 BWT_t BWT_Tree(int begin, int end, const string &T
         ){
17   if(end − begin <= THRESHOLD){
18     return BWT_Small(begin, end, T);
19   }
20   int mid = (begin + end)/2;
21   BWT_t left = BWT_Tree(begin, mid, T);
22   BWT_t right = BWT_Tree(mid, end, T);
23   return Merge(T, left, right);
24 }
```

図 **2** 提案手法の擬似コード
Fig. 2 Pseudo code of proposal method.

4. 提 案 手 法

本研究では分割統治法によるメモリ内での BWT の構築法を提案する．提案手法ではまずテキストを再帰的に分割し，ある程度のサイズになったところでその部分文字列の BWT を構築する．続いてそれらをツリー状にマージしていく．擬似コードを図 **2** に示す．

提案手法ではテキスト全体に対する SA を構築しないため，BWT を省メモリに構築できる．また，分割統治法を用いることでタスク並列モデルによる並列化が容易であるのに加え，マージ処理も並列化することで全体としての並列度を向上させることもできる．テキスト全体に対する SA を構築しないクリティカルパス $o(n)$ のアルゴリズムは我々の知る範囲では他にない．

4.1 部分文字列に対する BWT

図 2 の BWT_Small について説明する．まず部分文字列に対する SA を構築する．これには任意の文字列ソートアルゴリズムを用いることができる．部分文字列の長さを m とする．SA が得られれば，BWT は定義式より $O(m)$ で計算できる．

後に BWT をマージする際に SA が必要となるが，SA は $O(n \log n)$ ビットのメモリを必要とする．そこで，SA のサンプリングを行う．すなわち，部分文字列内の位置を一定間隔でサンプリングし，その位置に対する SA のみを保持する．SA のどの要素がサンプリングされているかはビット列 mark に記録する．すなわち，$SA[i]$ がサンプリングされていれば mark$[i] = 1$ とし，そうでなければ mark$[i] = 0$ とする．BWT，サンプリングされた SA，及び mark があれば，LF-Mapping によって SA の値を復元できる．サンプリング間隔を $\log n$ とすると，SA の復元には平均して $O(\log n)$ 回の LF-Mapping を要する．BWT，サンプリングされた SA，及びビット列 mark に必要なメモリはそれぞれ $O(n \log \sigma)$，$O(n)$，$O(n)$ ビットとなる．

図 **3** gap を用いた T_l^{bwt} と T_r^{bwt} のマージ
Fig. 3 Merging T_l^{bwt} and T_r^{bwt} using gap array.

4.2 BWT のマージ

次に，図 2 の Merge について説明する．ここでマージすべき対象は BWT，サンプリングされた SA，及びビット列 mark である．サンプリングされた SA と mark は BWT を再帰的にマージするために必要である．この 3 つの配列のマージはほぼ同様の方法で行えるため，以下では BWT のマージについて説明する．

マージの際に，左側・右側の部分文字列をそれぞれ T_l，T_r，それらに対する BWT をそれぞれ T_l^{bwt}，T_r^{bwt} とする．これらの長さは m であるとする．マージには Ferragina らの定理[4) を利用する．

定理 1.

$$T_l[SA[i], n] < T_r[t, n] < T_l[SA[i+1], n] \quad (3)$$

が成立しているとき，$T_r[t-1, n] = cT_r[t, n]$ の挿入位置が以下で表されるとする．

$$T_l[SA[j], n] < T_r[t-1, n] < T_l[SA[j+1], n] \quad (4)$$

このとき j は以下の式で求められる．

$$j = \begin{cases} C[c] + rank(c, i) + p & (\ c = T_l[m-1]) \\ C[c] + rank(c, i) & (otherwise) \end{cases} \quad (5)$$

ただし p は以下のように定める．

$$p = \begin{cases} 1 & (\ T_r[0, n] < T_r[i+1, n]) \\ 0 & (otherwise) \end{cases} \quad (6)$$

証明は Ferragina らの論文に譲るが，この定理によって，$T_r[m-1]$ から始まる suffix の挿入位置さえ分かれば，T_r^{bwt} の各要素がそれぞれ T_l^{bwt} のどこに挿入されるのかが帰納的に分かる．各位置に挿入される要素数をカウントした配列を以下では gap と呼ぶ．図 **3** に例を示す．gap にはそのままだと $O(n \log n)$ ビットのメモリが必要だが，要素の総和が T_r^{bwt} の要素数であるという性質を利用すると以下の定理が成り立つ．

定理 2. gap は $O(n \log \log n)$ ビットのメモリで表す

ことができる.

証明. T_r^{bwt} の要素数が一番大きくなるのは最後のマージのときであり, このときの要素数は $n/2$ である. gap の各要素に p ビットずつ割り当てると, 2^p-1 という値まで格納できる. ある要素の値が 2^p 以上になったときは, その要素の位置と値のペアにそれぞれ $\log n/2$ ビットを新たに割り当てる.

このとき, 追加領域を最大に用意しなければならないのは, gap の $(n/2)/2^p = n/2^{p+1}$ 個の要素が 2^p という値を持ち, それ以外の要素が 0 のときである. すると, 全体として確保するメモリ量は

$$f(p) = 2\frac{n}{2^{p+1}}\log\frac{n}{2} + \frac{n}{2}p \tag{7}$$

となる. $f(p)$ を微分して, 最小となるときの p を求めると以下のようになる.

$$f'(p) = -\ln 2\frac{n\log n/2}{2^p} + \frac{n}{2} = 0$$
$$p = O(\log\log n) \tag{8}$$

よって全体として必要なメモリは以下のようになる.

$$f(\log\log n) = O(n\log\log n) \tag{9}$$

□

$T_r[m-1]$ から始まる suffix の挿入位置を求めるには二分探索を行う. k を 2 つの suffix の辞書的な大小を決定するのに必要な平均比較回数とする. 文字列の二分探索の時間計算量は通常 $O(k\log n)$ であるが, 二分探索の各比較の度に, SA の値を 4.1 節で述べた方法で求める必要がある. LF-Mapping を高速に行うため, T_l^{bwt} は予め wavelet matrix[2] という形式に変換しておく. 詳細は割愛するが, これにより rank が $O(\log\sigma)$ の時間計算量で求まる. よって SA の要素の復元は平均して $O(\log\sigma\log n)$ の時間で行うことができ, 二分探索の時間計算量は $O(k\log\sigma\log^2 n)$ となる.

その後, T_r^{bwt} の各要素をそれぞれ T_l^{bwt} の要素の間に挿入する. これには式 5 を用いる. wavelet matrix によって rank は $O(\log\sigma)$ の時間で求まるので, 全ての要素を挿入するのに $O(n\log\sigma)$ の時間を要する.

4.3 LF-Mapping の修正

4.1, 4.2 節で述べた方法では, 各部分文字列が番兵を持たないため, LF-Mapping が正しく行われない. しかし, 予め各部分文字列に番兵を付加してしまうと suffix 間の辞書的な順序関係が保存されなくなる. そこで, 各部分文字列の BWT を構築したあとに擬似的に番兵を付加する必要がある. つまり, 正しい SA と BWT を求めた後に, SA の先頭に m を, BWT の先頭にその部分文字列の最後の文字を挿入する. また, 部分文字列の先頭から始まる suffix に対応する BWT の要素を$に置き換える. これにより, SA と BWT の値は正しいまま, 番兵の存在をシミュレートできる.

これだけだと L と F において同じ文字間の順番が保存されなくなるため, さらに LF-Mapping の修正が必要である. そのために, まず BWT の先頭に挿入された文字の F 中での正しい位置 s を求める. そして位置 i の LF-Mapping を以下のように変更する.

$$LF_{mod}(i) = \begin{cases} s & (i=0) \\ LF(i)-1 & (T_l^{bwt}[i] = T_l^{bwt}[0], \\ & \ LF(i) \le s) \\ LF(i) & (otherwise) \end{cases} \tag{10}$$

4.4 逐次アルゴリズムの計算量

提案手法では以上のようなマージをツリー状に行っていくことで, 最終的にテキスト全体に対する BWT を得る. BWT のマージの計算量は $O(n\log\sigma)$ であり, これを再帰の深さ分だけ行うので, 逐次アルゴリズムの時間計算量は $O(n\log\sigma\log n)$ となる. また, 空間計算量は $O(n\log\log n)$ となる.

4.5 並列化手法

提案手法は分割統治法に基づいており, タスク並列処理と相性がよい. すなわち, 提案手法では手続きを再帰的に呼び出しており, その再帰呼び出しを 1 つのタスクとみなすことで, 数多くのタスクを生み出すことができる. それらのタスクを各スレッド間で動的に分配することで, 並列化を実現できる. これによって, クリティカルパスは $O(n\log\sigma)$ にまで減少する.

マージに関しても並列化を行う. *gap* の構築では, T_r をさらに適当な大きさに分割し, その各ブロック内での最後から始まる suffix の挿入位置をそれぞれ二分探索によって求める. その後, それより前の suffix の挿入位置を順に計算するようにすれば, これはクリティカルパス $O(k\log\sigma\log^2 n)$ の処理になる. そこからの BWT のマージは *gap* に対する prefix sum を求められれば並列に行うことができるため, クリティカルパスは $O(\log n)$ である. また, wavelet matrix の構築も並列に行う必要がある. 詳しくは述べないが, これは $O(\log\sigma\log n)$ で行うことができる.

これらの処理が再帰の深さ分だけあるので, アルゴリズム全体のクリティカルパスは $O(k\log\sigma\log^3 n)$ となる. 2 つの suffix の比較が長くなってしまう場合には, それを防ぐためのデータ構造を予め構築しておくことで, 比較回数を減らすことができる. 例えば Kärkkäinen の difference cover sample[5] を用いれば,

ⓒ 2013 Information Processing Society of Japan

k の値を $O(\log^2 n)$ に抑えることができ, 全体のクリティカルパスは $O(\log \sigma \log^5 n)$ となる.

5. 評　価

提案手法の評価として, まずデータサイズを変えたときの逐次アルゴリズムの実行時間とメモリ使用量の変化を調べた. メモリ使用量は getrusage システムコールによって測定した. 比較対象として, テキスト全体の SA を SA-IS によって構築してから BWT を計算する方法, 及び岡野原らの手法である BWT-IS を使用した. 入力にはランダム英数字で構成されるテキストを用いた. 評価は CPU が Intel(R) Xeon(R) E7540 2.00GHz, メモリ 256GB のマシンで行った. コア数は 24 である. 評価は全て単一ノード上で行った.

結果を表 **1**, 表 **2** に示す. 提案手法の逐次アルゴリズムは他の 2 つと比べて時間計算量が大きいため, 実行時間が大きい. メモリ消費に関しては, BWT-IS よりも抑えられている. しかし, 現段階の提案手法の実装は繰り返しが多いテキストに弱い等の弱点を持っており, 純粋に BWT-IS より優っているとは言えない. そのため, 今後改めて比較を行う予定である.

次に, 提案手法のスケーラビリティを測定したものを図 **4** に示す. 並列化には我々の開発しているライブラリである MassiveThreads を利用した. テキストは 100MB のものを使用した. 図中の値は 1 コアで実行した場合に対する相対性能を計算したものである. 並列化により性能が最大で 22.69 倍にまで向上した. このときの実行時間は 205.09 秒であり, これは BWT-IS の性能を上回っている.

6. お わ り に

本研究では文字列検索のための圧縮インデックスの省メモリかつ並列化可能な手法を提案した. 評価によりメモリ使用量の削減を確認し, スケーラビリティも良好であることが確かめられた. 今後の展望としては, 繰り返しが多く現れるテキストへの対処と, さらなる性能の向上を考えている.

表 **1**　データサイズに対する実行時間 [s] の変化
Table 1　Change of execution time[s] to data size.

	proposal	BWT-IS	SA-IS
10[MB]	358.82	8.89	4.53
20[MB]	772.02	23.76	13.80
50[MB]	2135.24	80.45	53.83
100[MB]	4652.87	209.35	126.50

表 **2**　データサイズに対するメモリ使用量 [MB] の変化
Table 2　Change of memory usage[MB] to data size.

	proposal	BWT-IS	SAIS
10[MB]	56.21	80.45	98.67
20[MB]	101.61	147.14	194.78
50[MB]	245.34	299.29	479.42
100[MB]	437.68	532.21	901.14

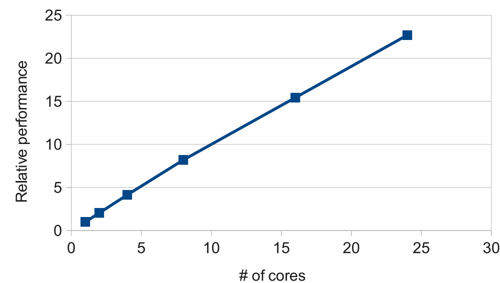

図 **4**　1 コアの性能を 1 とした時の相対性能の変化
Fig. 4　Relative performance to 1 core execution.

参 考 文 献

1) Burrows, M., Wheeler, D. J., Burrows, M. and Wheeler, D. J.: A block sorting lossless data compression algorithm, Technical report (1994).
2) Claude, F. and Navarro, G.: The Wavelet Matrix, *Proc. SPIRE'12*, pp. 167–179 (2012).
3) Ferragina, P. and Manzini, G.: Opportunistic data structures with applications, *FOCS '00*, pp. 390–398 (2000).
4) Gagie, P. F. T. and Manzini, G.: Lightweight Data Indexing and Compression in External Memory, *Algorithmica*, Vol. 63, No. 3, pp. 707–730 (2012).
5) Kärkkäinen, J.: Fast BWT in small space by blockwise suffix sorting, *Theoretical Computer Science*, Vol. 387, No. 3, pp. 249–257 (2007).
6) Kulla, F. and Sanders, P.: Scalable parallel suffix array construction, *Parallel Computing*, Vol. 33, pp. 605–612 (2007).
7) Manber, U. and Myers, G.: Suffix arrays: A new method for on-line string searches, *SODA '90*, pp. 319–327 (1990).
8) Nong, G., Zhang, S. and Chan, W. H.: Linear Suffix Array Construction by Almost Pure Induced-Sorting, *Data Compression Conference*, pp. 193–202 (2009).
9) Okanohara, D. and Sadakane, K.: A Linear-Time Burrows-Wheeler Transform Using Induced Sorting, *SPIRE '09*, pp. 90–101 (2009).
10) 岡野原大輔: 高速文字列解析の世界, 岩波書店 (2012).

ⓒ 2013 Information Processing Society of Japan